Increasing Demultiplexing Efficiency in TCP/IP Network Servers[1]

Joseph T. Dixon and Kenneth L. Calvert
{jdixon,calvert}@cc.gatech.edu

Networking and Telecommunications Group
College of Computing
Georgia Institute of Technology

Abstract

This paper shows how software caches and hashing to multiple PCB (protocol control block) lists can increase demultiplexing efficiency in TCP/IP network server. We implemented six algorithms and executed 200 simulations -- using four server traffic traces as input -- to formulate best-use caching and hashing policies for demultiplexing TCP-based http, telnet and login services and UDP-based services such as nfs. We recommend several server-independent modifications that can yield potentially significant demultiplexing performance benefits.

1. Introduction

As network services (such as *World-Wide Web*, connection services, etc.) proliferate, greater performance demands are placed on the computers that provide them. Likewise, recent efforts have identified packet processing rather than link bandwidth as the primary bottleneck in today's high-speed networks [Klein95]. Thus, network servers must process packets with optimal efficiency. In this paper, we quantify the performance benefits of different organization and search strategies for locating state information relevant to incoming packets -- an important component of packet processing in the TCP/IP suite.

In TCP/IP, packets arriving to a server host must ultimately be delivered to the kernel level incarnation (i.e., the protocol control block, or PCB) of a higher layer process. The proper PCB is determined by the source and destination IP addresses and port numbers in the packet; this is called *demultiplexing* the packet. TCP and UDP demultiplexing are often implemented as a linear search of a linked list of PCBs until a best-match is found. A linear search, though simple, can be quite inefficient when a large number of PCBs are referenced multiple times in a sufficiently random order.

[1] The extended version of this paper is available at: ftp.cc.gatech.edu/pub/coc/tech_reports/1996/GIT-CC-96-08.ps.Z.

Two simple observations have led to implementations that improve performance. First, Clark showed that often, once a PCB (or small set of them) is referenced, it tends to be preferentially referenced in the future [Clark89]. In this case, a single cached PCB pointer can yield substantial performance gains. This pointer always directly addresses the last PCB referenced; thus, *a cache hit* allows referencing of the correct PCB without the costly list search. Second, McKenney and Dove showed analytically that, under certain conditions, improved performance can be achieved when PCBs are distributed over several *hash chains* [McK&Dove92]. A hash function determines the hash chain to search.

We constructed trace-driven simulations that characterize both caches and hash chains as performance enhancers. Six demultiplexing algorithms were implemented:

- a simple linear search of a linked list of PCBs
- the single-entry cache algorithm of BSD 4.3-Reno
- two and three-entry cache extensions of the BSD 4.3-Reno algorithm
- two parameterized algorithms combine caching and hash chains

Subsequent sections of this paper present the details of our work, show our experimental results, and present important recommendations for server demultiplexing implementations.

2. Related Work

A number of researchers have investigated demultiplexing efficiency; our efforts differ in several important ways: (1) we establish clear boundaries on the best use of caching alone and in combination with hashing; (2) our analysis is based on traces of packets arriving at actual network servers; (3) we make several concrete recommendations for simple, server-independent modifications. The most relevant works are:

- Clark, et. al., suggested that caching a pointer to the last referenced TCB (transmission control block) yields a substantial performance benefit [Clark89].
- McKenney and Dove showed combining caching with multiple hash chains is "better" than other well-known alternatives for on-line transaction processing systems [McK&Dove92]..
- Mogul investigated persistence and temporal locality at the process level and showed that arriving packets are often destined for the process that most recently sent a packet, and often with little intervening delay [Mogul92].
- Kay and Pasquale measured the cost of packet multiplexing and demultiplexing as part of overall non-data touching overhead for systems that send/receive small packets [Kay&Pasq93].
- Partridge and Pink found that a single-entry cache had little effect on UDP lookups [Part&Pink93].

3. Overview of Network Traces

Our analysis is driven by traces collected on four (4) Georgia Institute of Technology (GIT) network servers from 4:35PM to 6:19PM, April 7, 1995 (during peak usage) using the UNIX `tcpdump(1)` command. The servers were directly attached to the College of Computing's 100Mbps FDDI backbone and provided a variety of TCP and UDP services to both on and off-campus hosts. Table 1 shows various high-level characteristics of the traces.

Server Name	Primary Services	Total Incoming Packets	Total TCP Lookups	Total UDP Lookups
cleon	login, mail, telnet	183462	83879	85975
gaia	login, telnet, xterm	190564	60569	106969
lennon	login, nfs, smtp, telnet	174949	110497	61437
siwenna	http, nfs, ntp, ftp	1222643	328379	143830

Table 1: Summary of the four network server traces and number of PCB lookups for each.

4. TCP and UDP Demultiplexing Algorithms

Traditionally, `in_pcblookup()` is the kernel function that performs the PCB list search. This function matches the source and destination IP addresses and port numbers of a just-arrived packet (represented below as $<I_1,P_1><I_2,P_2>$) with the foreign/local addresses of a PCB; the latter may include "wildcard" addresses, indicating that the PCB can accept packets with any value in that field. The only mandatory value specified in a wildcarded PCB is the local port number; any combination of foreign IP address, foreign port and local IP address may be unspecified. When an inbound packet is received, `in_pcblookup()` searches through a linked list of PCB's (a pointer to which is passed as an argument) and identifies the PCB with the fewest wildcard matches. A NULL value is returned when no PCB can be found. All the algorithms implemented for this study eventually invoke `in_pcblookup()`. We measured its performance to serve as a benchmark (since it was the original demultiplexing solution).

4.1 1-, 2-, and 3-entry Caching of Last Referenced PCB(s)

The BSD 4.3-Reno UNIX release was the first to include a single cached pointer to the last PCB referenced. If the next packet that arrives is destined for the PCB that the cache identifies, then a list search is avoided; otherwise, a call to `in_pcblookup()` is needed to find the best match PCB.

We devised two- and three-cache algorithms that extend the principles of the BSD 4.3-Reno algorithm. Though conceptually simple, these algorithms have an inherent complexity: more than one element in the cache requires policies governing access order and replacement. Our implementations of two- and three-cache algorithms enforce *most-recently-used* (MRU) access and *least-recently-used* (LRU) replacement policies. Both policies capitalize on expected high locality of reference within the PCB list. Figure 1 shows the structures used in the 2-cache algorithm after packets destined for PCBs $<I_1,P_1><I_2,P_2>$ and $<I_3,P_3><I_4,P_4>$ are demultiplexed.

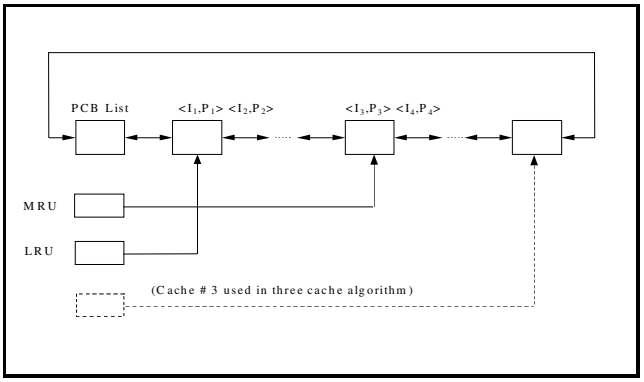

Figure 1: Diagram of 2- and 3-cache demultiplexing algorithms.

The 2-cache algorithm executes the following steps when a received packet is demultiplexed.

```
Check the MRU cache;
If !(MRU cache hit) {
        Check the LRU cache;
        If !(LRU cache hit)
                LRU cache =
                in_pcblookup(PCB_list, ...);
        swap(MRU cache, LRU cache);
        }
Return MRU cache as the destination PCB;
```

This algorithm accesses the MRU cache first, but replaces it last; it accesses the LRU cache last, but replaces it first. The three-cache version is analogous, except that a third "intermediate" cached PCB pointer must be managed along with the MRU and LRU caches. As new PCBs are cached in MRU, the old contents are first rotated to the intermediate cache, then to the LRU cache, and so on.

4.2 Combining Caching and Hashing.

McKenney and Dove introduced a demultiplexing algorithm that combines software caching and multiple hash chains. The algorithm maintains a linear list of PCBs for each of several hash chains. Each hash chain has an associated cache that points to the last PCB found on that chain. When a packet arrives, it is routed to a hash chain via a hash function. The packet is assigned to its PCB via a BSD 4.3-Reno type search of the list. Figure 2 shows the structures used in this algorithm after

packet destined for PCBs <I_1,P_1><I_2,P_2> and <I_3,P_3><I_4,P_4> are demultiplexed.

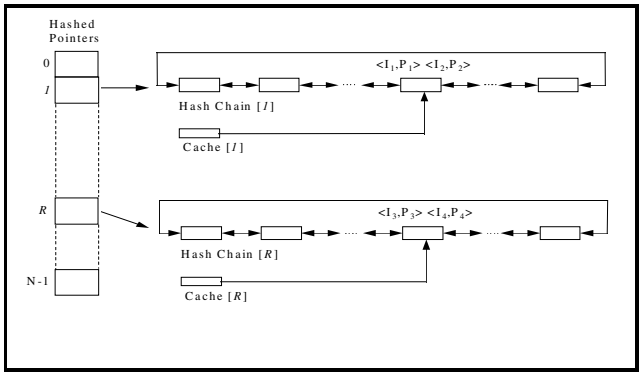

Figure 2: Diagram of an *N* hash chain algorithm with 1 cache per chain.

The 1-cache/multiple hash chain algorithm we used executes the following steps when a received packet identified by <I_1,P_1><I_2,P_2> is demultiplexed.

```
Compute R = hash(<I1,P1><I2,P2>), where R ∈ [0,(N-1)];
Check cache[R];
If !(cache hit)
        cache[R] =
        in_pcblookup(hash_chain[R], ...);
Return cache[R] as the destination PCB;
```

5. Methodology and Metrics

We measure performance in terms of assembly language instructions required per lookup. We used the following procedure to determine the number of instructions executed by each algorithm:

1. Implement a *C*-language version of the algorithm.
2. Generate an optimized assembly language version of the *C* program using the *gcc* compiler.
3. Determine the correct mapping between the assembly language instructions and the *C* program instructions. (This step determines the exact cost of each logical processing path.)
4. Imbed the code to compute instruction counts for each lookup (from step 3) in the *C* program.

We adopted several implementation conventions that make our results more realistic and we made several experimental assumptions to overcome limitations (in particular, lack of information about TCP header flags) of the packet traces.

- For algorithms that use a PCB cache, wildcarded PCBs are cached when they are the best match.
- New PCBs are added to the front of the PCB list(s).
- All calls to in_pcblookup() allow for wildcard matches in the specified list.
- We used a *maximum segment lifetime* value of 60 seconds in our simulations, to determine when a PCB would be removed from the list (i.e. 120 sec after the last packet of the connection).
- Appropriate wildcard PCBs were included for the services offered by each server for each simulation.
- The first packet of each connection is treated as a connection request when a TCP packet arrives that has a *well-known* local port and a *non-well-known* foreign port.
- A fully specified PCB is added to the UDP list when a UDP packet arrives with a *non-well-known* local port.

6. Results for TCP-based Services

6.1 PCB caching improves efficiency.

Our results corroborate the findings of several past efforts: although cache management cost is added to every TCP PCB lookup, the BSD 4.3-Reno cache capitalizes on the high locality of reference of TCP lookups, so that many long list searches are avoided. That the BSD 4.3-Reno algorithm provided a substantial performance gain was no surprise. However, if a single cache results in such benefits, how might additional software caches affect performance? A popular presumption asserts that there is little or no benefit (mainly due to cache management overhead) in using more than one software cache to directly reference PCBs for incoming packets (e. g., [Part&Pink93]). Table 2 compares the mean number of instructions executed per TCP PCB lookup for the in_pcblookup (), BSD 4.3-Reno algorithm, and the 2-cache algorithms. The 2-cache algorithm shows that a two element cache can be managed efficiently enough so the benefits (i. e., avoided PCB list searches) outweigh the costs.

Server	in_pcb lookup	BSD 4.3-Reno	2-cache	Benefit over in_pcb lookup	Benefit over BSD 4.3-Reno
cleon	132.47	49.76	32.45	75.5 %	34.7 %
gaia	202.73	112.01	72.5	64.2 %	35.8 %
lennon	368.27	229.72	139.47	62.1 %	39.3 %
siwenna	416.34	337.47	289.1	30.5%	14.3 %

Table 2: Mean of assembly language instructions executed per TCP PCB lookup for in_pcblookup(), BSD 4.3-Reno, and 2-cache algorithm.

In all four servers, a larger fraction of TCP lookups are performed at a lower instruction execution cost. For example, consider the 0.80-quantile for each algorithm presented thus far:

- cleon improves from 180 to 64 to 29 instructions;
- gaia from 288 to 201 to 101 instructions;
- lennon from 675 to 439 to 233 instructions;
- siwenna from 648 to 559 to 491 instructions.

6.2 Benefits diminish with more than two cache entries.

Clearly, whether performance gains can be realized hinges on whether cache hit benefit can exceed cache management cost. Our results show that performance gains diminish for our servers when three cache entries are used rather than two. Table 3 shows our results.

Server	2-cache	3-cache	% Degradation
cleon	32.45	54.72	-68.6 %
gaia	72.5	110.4	-52.2 %
lennon	139.47	207.82	-49.0 %
siwenna	289.1	317.71	-9.8 %

Table 3: Mean number of assembly language instructions executed per TCP PCB lookup for the two- and three-cache algorithms.

6.3 Caching alone does not reduce demultiplexing cost variability.

For some types of services (e. g., real-time services such as video), variability in lookup time is as important as overall delay. Table 4 shows that standard deviation for the number of assembly language instructions executed changes little with the caching algorithms.

Algorithm	cleon	gaia	lennon	siwenna
in_pcblookup	65.67	145.96	277.34	439.74
BSD 4.3-Reno	64.89	137.81	270.24	447.73
2-cache	41.22	109.86	220.93	440.75
3-cache	67.03	136.75	263.56	443.09

Table 4: Standard deviation for number of assembly language instructions executed per TCP PCB lookup.

6.4 Combining caching with hash chains reduces cost and variability.

We implemented a 1-cache/128-hash chain algorithm using two trivial hash functions and managing wildcard PCBs on a separate list. We found significant decreases in both mean and standard deviation of cost. Our hash function was a simple *foreign port-modulo-N* (N is the number of hash chains), which we found to distribute PCBs across hash chains as effectively as *Fletcher* or *Xor-folding* hashes. In addition, we found that managing wildcard PCBs on a separate list resulted in negligible impact on performance. Our simulation results are presented in Tables 5 and 6. We also implemented a 2-cache, multiple hash chain algorithm, but found that performance was only marginally better (when the number of hash chains is small.)

Server	in_pcblookup()	Hash chains with caching	% Improvement
cleon	132.47	23.98	81.8 %
gaia	202.73	27.93	86.2 %
lennon	368.27	35.29	90.4 %
siwenna	416.34	28.97	93.0 %

Table 5: Mean number of assembly language instructions executed per TCP PCB lookup for a direct call to in_pcblookup() and the 1-cache/128-hash chain algorithm.

Server	in_pcblookup()	Hash chains with caching	% Improvement
cleon	65.67	16.76	74.4 %
gaia	145.96	27.96	80.0 %
lennon	277.34	61.73	77.7 %
siwenna	439.74	193.15	56.0 %

Table 6: Standard deviation of number of assembly language instructions executed per TCP PCB lookup for a direct call to in_pcblookup() and the 1-cache/128-hash chain algorithm.

Figures 3(a)-(d) show the cumulative distribution of the demultiplexing algorithms discussed thus far.

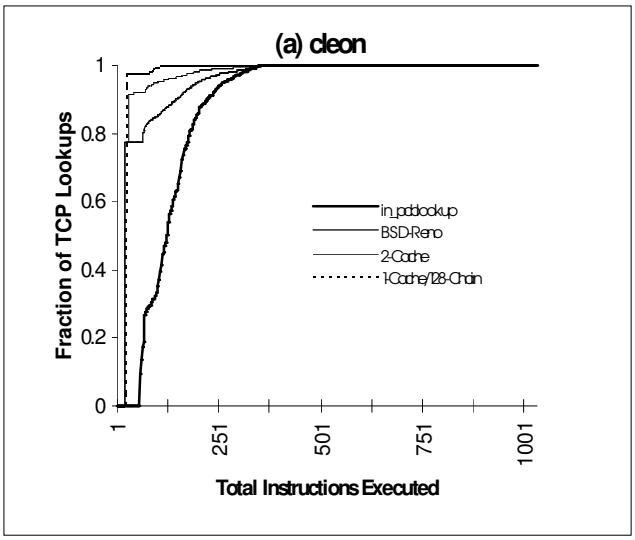

Figure 3(a): Cumulative distribution of instructions executed for TCP lookups on the *cleon* server.

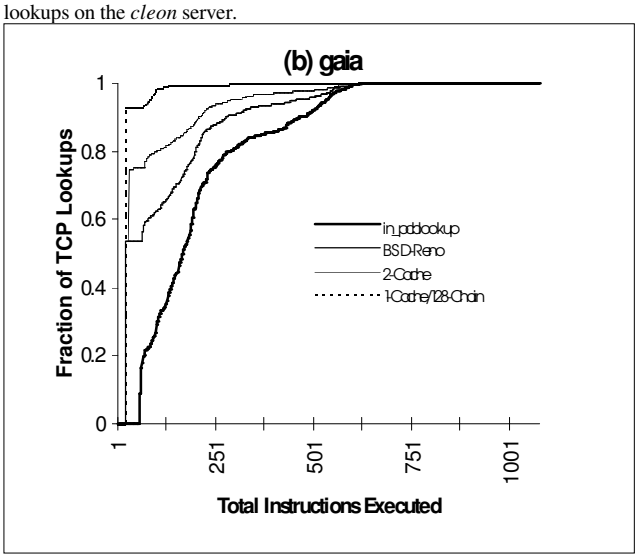

Figure 3(b): Cumulative distribution of instructions executed for TCP lookups on the *gaia* server.

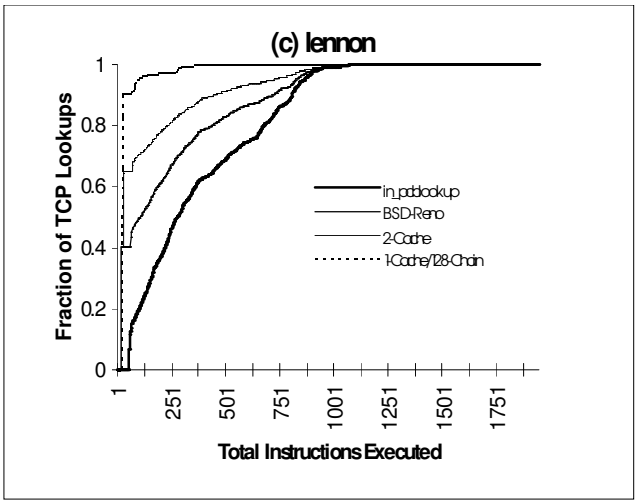

Figure 3(c): Cumulative distribution of instructions executed for TCP lookups on the *lennon* server.

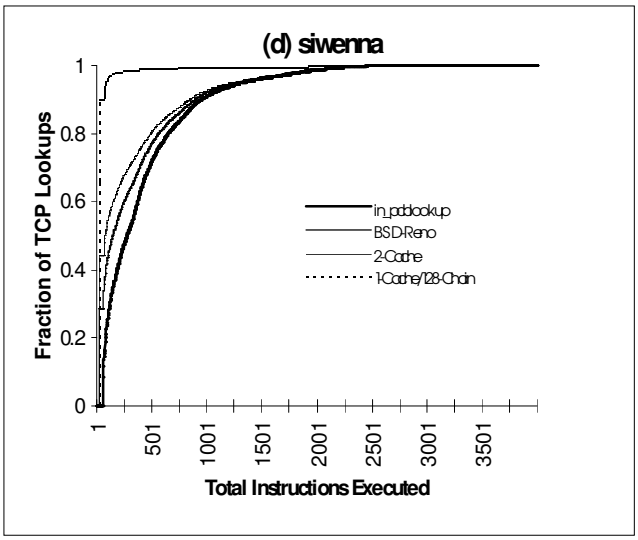

Figure 3(d) CDF of instructions executed for TCP lookups on the *siwenna* server.

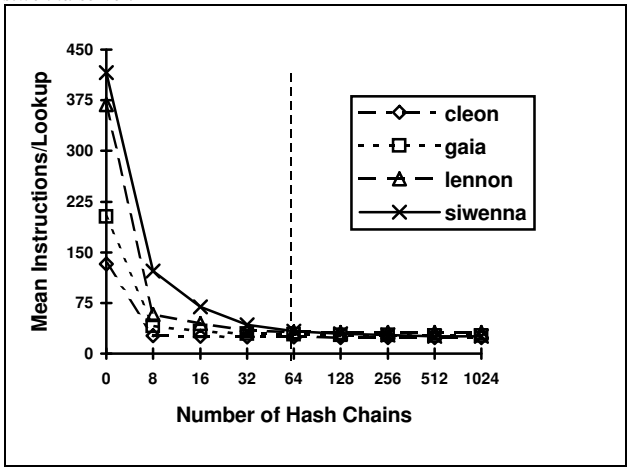

Figure 4: Mean number of assembly language instructions executed per TCP PCB lookup for single 1-cache/hash chain algorithm for various numbers of hash chains.

6.5 A small number of hash chains provide the majority of possible performance gains.

An obvious issue related to use of hash chains in demultiplexing is how to choose the number of chains to attain the greatest performance gain. Figure 4 shows that the mean number of instructions per lookup continually decreases (asymptotic at 22 instructions per lookup) as the number of hash chains increase. However, its most significant feature is that, for our traces, there is little additional benefit beyond 64 hash chains.

7. Results for UDP-based Services

No kind of caching-only solution assures performance improvement on our servers' UDP traffic. A direct call to in_pcblookup() performed as well as or better than one, two- and three-cache algorithms for all our simulations. The 1-cache/128-hash chain and 2-cache/128-hash chain algorithms were more efficient than a direct call to in_pcblookup() for only two of the four network server traces. In those two cases (lennon and siwenna), instruction count mean and standard deviation were reduced substantially, although not to the levels we saw with TCP-based servers. Table 7 shows these results.

Number of Hash Chains	cleon	gaia	lennon	siwenna
in_pcblookup()	*211.89*	*1016.16*	*1013.80*	*807.41*
1-cache/128-hash chain	241.36	1055.06	394.39	353.81

Table 7: Mean number of assembly language instructions executed per UDP PCB lookup for the 2-cache/128-hash chain algorithm.

With most UDP-based services, a separate PCB is *not* created for each new client, and therefore more packets go to wildcarded PCBs. Most of the savings are thus due to reduced list searches on cache misses.

8. Combined Demultiplexing Results

8.1 Performance may be no better than in_pcblookup() when the same algorithm is used for TCP and UDP.

The fraction of UDP lookups to total lookups (UDP and TCP) may be sufficiently large so that inefficiency in UDP demultiplexing overwhelms efficiency gains in TCP demultiplexing. For example, the 0.80-quantile performance of the more sophisticated algorithms in 10 out of 12 simulations is no better than linear search. Figures 5(a) and 5(b) show the two cases for total server demultiplexing in which in_pcblookup() (applied to

both TCP and UDP lookups) outperformed all the other algorithms.

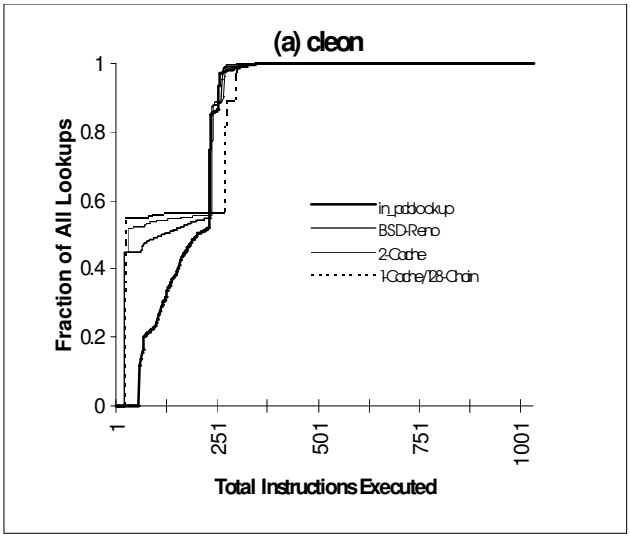

Figure 5(a): Cumulative distribution of instructions executed for all lookups on the *cleon* server.

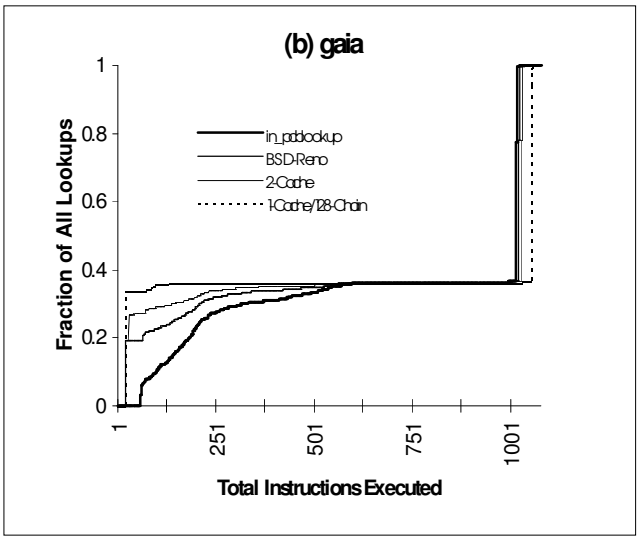

Figure5(b): Cumulative distribution of instructions executed for all lookups on the *gaia* server.

8.2 Separable demux algorithms improves efficiency for some servers.

If a separable demultiplexing solution is used, then the best algorithm can be chosen for each. Figure 6(a) and 6(b) compares cumulative distributions to illustrate how a separable solution can be used improve overall server performance. It compares a unified approach (both UDP and TCP use the 1-cache/128-hash chain algorithm) and a separable solution (combined cache/hash chain algorithm for TCP, direct call to `in_pcblookup()` for UDP) for our cleon and gaia network servers. Using this method, the mean instructions required for a lookup was lowered from 135.7 to 120.7, or 11%, on cleon and from 686.1 to 661.22, or about 4%, on gaia.

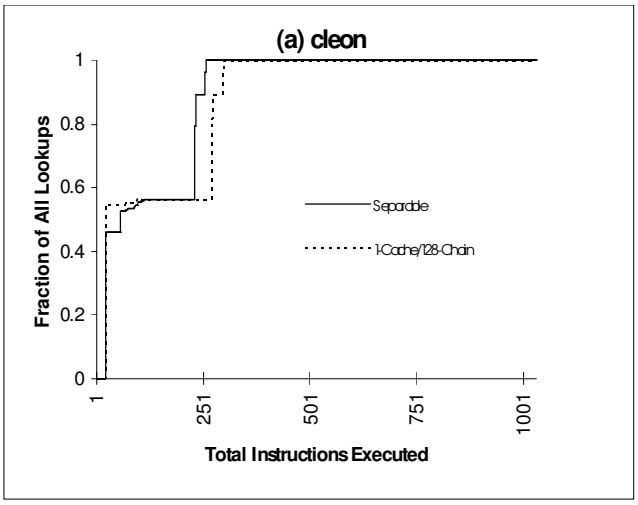

Figure 6(a) The CDFs of instructions executed for all lookups on *cleon* show the benefit of a separable solution.

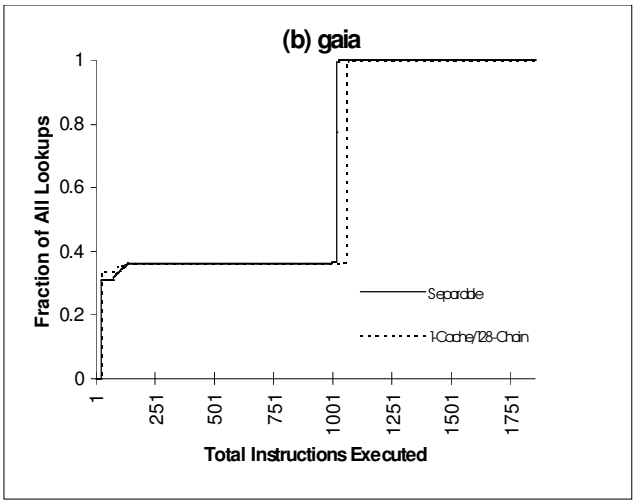

Figure 6(b) The CDFs of instructions executed for all lookups on *gaia* show the benefit of a separable solution.

9. Recommendations

As a result of our study, we formulated a set of recommendations, featuring modifications that can be implemented on a per server basis.

9.1 Use an algorithm that combines software caching with multiple hash chains on TCP-based services.

The concurrent nature of most TCP-based server programs makes software caching/multiple hash chain demultiplexing ideal. Our findings clarify three critical issues in implementing such an algorithm.

1. Wildcard PCBs should be managed on a separate list -- Wildcard matches are best handled when a failed match in the fully specified PCB hash chain is followed by a search of a wildcard PCB list.
2. The hash function should be trivial -- Foreign port number or IP address *modulo* the number of hash

chains provides substantial performance gains, depending on traffic mix.

3. The number of hash chains need not be large -- The vast majority of performance gain is realized by having roughly one hash chain for one to two clients. As the number of hash chains is increased beyond this, the relative performance gain is marginal.

9.2 If no single service dominates incoming UDP traffic, use a direct call to in_pcblookup() .

If a network server offers many UDP-based services, then attempts to improve demultiplexing performance with caching or hash chains may be fruitless. Most UDP-based servers operate on a request-response model; if a server has a uniform mix of traffic, caching and hash chain benefits may be overwhelmed by many list searches.

9.3 Use a caching/multiple hash chain algorithm when a single service makes up the majority of incoming UDP traffic.

In some cases, most incoming UDP traffic is destined for a single service, or small set of them. If so, then a caching/multiple hash chain algorithm offers benefits arising from reduced cost of list searches on cache misses. Our servers, whose incoming UDP traffic was dominated by the *NFS* service, showed significant UDP demultiplexing performance gains.

9.4 Make TCP and UDP packet demultiplexing separable.

Our experiments clearly show that, while an algorithm might yield exceptional performance gains for demultiplexing TCP packets, using the same algorithm for UDP packets can add unnecessary overhead. The best performance can often be achieved when different algorithms can be used for TCP and UDP. This was the case for two of our servers.

10. Conclusions

Our work has focused on improving efficiency within the traditional TCP and UDP demux structure. Recent proposals for less traditional approaches include passing 32-bit PCB identification parameters as a TCP connect-time option [Huitema95], and *source hashing*, which allows direct access to various information associated with general packet processing [Chan&Varg95]. In Mentat, Inc.'s TCP/IP implementation, for example, incoming packets are demultiplexed by IP rather than TCP or UDP [Mentat93]. The long-term solution for demultiplexing is not clear. Meanwhile, our conclusions and recommendations provide simple, server-independent solutions that yield performance gains at such high levels that, for some servers at least, additional enhancements may be unnecessary.

Acknowledgement - The authors thank Dan Forsyth for assisting with collecting the packet traces.

11. References

[Chan&Varg95] Girish P. Chandranmenon and George Varghese, Trading packet Headers for Packet Processing, SIGCOMM '95, 1995.

[Clark89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen, An Analysis of TCP Processing Overhead, IEEE Communications Magazine, June, 1989.

[Comer94] Douglas E. Comer and David L. Stevens, Internetworking with TCP/IP: Client-Server Programming and Applications, Prentice Hall, Inc., 1994.

[Huitema95] Christian Huitema, Multi-homed TCP - IETF Draft, Network Working Group, May, 1995. *This is a work in progress.*

[Jain91] Raj Jain, A Comparison of Hashing Schemes for Address Lookup in Computer Networks, IEEE, June, 1991.

[Kay&Pasq93] Jonathan Kay and Joseph Pasquale, The Importance of Non-Data Touching Processing Overheads in TCP/IP, ACM SIGCOMM '93, September, 1993.

[Klein95] Karl Kleinpaste, Peter Steenkiste, and Brian Zill, Software Support for Outboard Buffering and Checksumming, SIGCOMM '95, 1995.

[McK&Dove92] Paul E. McKenney and Ken F. Dove, Efficient Demultiplexing of Incoming TCP Packets, ACM SIGCOMM '92, August, 1992.

[Mentat93] Mentat TCP/IP Design Overview (extracted from Mentat TCP/IP Internals Manual), Mentat, Inc., Los Angeles, CA., July, 1993.

[Mogul92] Jeffrey C. Mogul, Network Locality at the Scale of Processes, ACM Transactions on Computer Systems, May, 1992.

[Part&Pink93] Craig Partridge and Stephen Pink, A Faster UDP, IEEE/ACM Transactions on Networking, July, 1993.

[Stevens94] W. Richard Stevens, TCP/IP Illustrated, Volume 1 -- The Protocols, Addison-Wesley Publishing Company, 1994.

[Wright94] Gary R. Wright and W. Richard Stevens, TCP/IP Illustrated, Volume 2: The Implementation, Addison-Wesley Publishing Company, 1995.